SFA Modernization Program
United States Department of Education
Student Financial Assistance

Java Coding Standards

Task Order #16
Deliverable #16.1.3

July 7, 2000

Table of Contents

1 Introduction

1.1 Purpose

The Java Coding Standards document will assist SFA in developing Java programs in a uniform, consistent manner. This coding standard document will ensure that programs developed under these guidelines will be readable, maintainable by different programmers, and easier to debug and test.

1.2 Scope

This document covers Java coding standards and includes:

- Java Overview – A high level overview of Java.
- Source Files – Organization of the source file.
- Code Layout – Structure the code should be written in.
- Naming Conventions – Standard methods of labelling.
- Coding Priorities – High-level requirements that are to be observed during coding and their definitions.
- Programming Style – Layout conventions and principles used when coding.
- Comments – Guidelines for commenting code to explain functionality.
- Code Example – An example of source code that may be used as a reference to the correct adherence of the principles and guidelines outlined within the document.
- There will also be examples of code to support the understanding of the principles/guidelines. The examples of code that follow the ***incorrect adherence*** of the principle/ guideline may still function correctly, but still do not follow the ***correct adherence*** to the principle/guideline stated.

Note: Deliverable 4.1.5 – Internet Standards is included as an appendix to this document. This deliverable contains supplementary information that is relevant to an overall Java application development strategy. Specifically:

- Enterprise JavaBeans Modeling
- Java Server Pages Views
- Java Application Architecture
- Java Applet Construction Standards
- Browser-Based Application Construction Standards

1.3 Audience

This coding standards document is aimed at an audience of technical architects, designers and developers charged with creating Java-based solutions. This document assumes that the reader is knowledgeable in Java coding and Java-based solutions.

2 Java Overview

Java is a programming language designed for use in the distributed environment of the Internet. It's design is based upon the C++ language. The current SFA Java code standard is Java 2 Platform Enterprise Edition (J2EE). Java supports many programming features which make it a powerful and flexible language for building applications.

The main features are:

- Portable - The Java Virtual Machine (JVM) interprets Java coded applications and enables Java code to be platform independent. As long as the JVM is integrated into the operating systems, any Java applications can be executed without having to be recompiled.

- Secure - When Java programs are downloaded and run on the JVM, Java supports a flexible, fine-granulated access system which grants authorized access only.

- Robust - Java has the ability to de-allocate memory to prevent bottlenecks and program crashes. Java also has ability of exception handling abilities to deal with errors and recovery.

- Object Oriented - Java allows the development of applications through a set of self-contained objects that interact. Each object encapsulates its data and exposes methods to other objects.

- Multithreading - By means of threads, independent execution paths are built. These threads are built at the language level, enabling uniform support and APIs for multithreading operations across all Java supported platforms.

- Dynamic - Required program components (classes) can be loaded from either the network or from local memory, enabling Java programs to adapt to its environment.

Note: Java Coding Standards may differ based upon the development tool used.

3 Source Files

A source file should contain one public class or interface. When private classes and interfaces are associated with a public class, they can be placed in the same source file as the public class. The public class should be the first class or interface in a source file.

Java source files are organized in the following manner:

- Beginning comments.
- Package Statement.
- Import statements.
- Class and interface declarations.

3.1 Beginning Comments

Beginning comments serve as a file header for the source file and provide information on the archive, revision, date, and author.

EXAMPLE:

```
/*
 * $Archive: $
 * $Revision: $
 * $Date: $
 * $Author: $
 */
```

3.2 Package Statement

A package statement is the first non-comment line of a Java source file. All custom-developed packages will utilize `gov.sfa` as their naming base (Refer to Deliverable 4.1.5 – Internet Standards in the appendix for additional information).

EXAMPLE:

```
package java.awt;
package gov.sfa.schools;
```

3.3 Import Statements

In the import section, it is recommended that each imported module is listed explicitly.

EXAMPLE:

CORRECT

```
import java.awt.Frame;
import java.awt.Graphics;

import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import java.applet.AppletContext;
```

INCORRECT

```
import java.awt.*;
import java.awt.event.*;
import java.applet.*;
```

3.4 Class and Interface Organization

The elements of a class should be in the following order:

- Class header.
- Constants (`final` class variables): `public, protected, private.`
- Public static inner classes.
- Protected inner classes, `static` or otherwise.
- Private inner classes, `static` or otherwise.
- Class variables (`private` only).
- Constructors.
- Other methods:

  When ordering methods, ignore visibility specifiers (`public, protected, private`) and follow these guidelines:

  - Keep related methods together.
  - When overriding superclass functions, keep them in the same order as in the superclass, and preferably together.
  - The class should end with the `unitTest`, `getExpectedResult` and main methods.

4 Code Layout

The recommendations in this section are designed to create a solid code layout strategy. This strategy will represent the logical structure of the code, make the code readable, and easy to maintain.

4.1 Class Headers

- Write class headers on a single line (if the line does not exceed 80 characters), when possible.
- If not, break the line before `extends` and `implements`. Indent succeeding lines.
- If the class header is on a single line, put the opening brace at the end of that line.
- If the class header needs multiple lines, put the opening brace left aligned on a line by itself.

4.2 Method Headers

- Write method headers on a single line (if the line does not exceed 80 characters), when possible.
- If not, break the line immediately after the opening parenthesis. This leaves all the parameters on the same line.
- If the method header is on a single line, put the opening brace at the end of that line.
- If the method header needs multiple lines, put the opening brace left aligned on a line by itself.

4.3 Indentation

When indentation is necessary, use four spaces.

EXAMPLE:

CORRECT

```
if (bottom < index) {
    this.topRow = index – this.rows;
} else if (index < this.topRow) {
    this.topRow = index;
}
```

INCORRECT

```
if (bottom < index)
  this.topRow = index – this.rows;
else if (index < this.topRow)
 this.topRow = index;
```

4.4 White Space in the Code

Blank lines improve readability by setting off sections of code that are logically related.

Add two blank lines in the following places:

- Between sections of a source file.
- Between class and interface definitions.

One blank line should be used in the following circumstances:

- Between methods
- Between the local variables in a method and its first statement
- Before a block (see section **8.1.1**) or single-line (see section **8.1.2**) comment
- Between logical sections inside a method to improve readability

Add one space in the following places:

- Between operators.
- After semicolons in `for`-loops.
- Before and after the assignment operator.
- *No* space in the following places:
  - Between a method name and the opening parenthesis.
  - Around opening and closing parentheses in a function declaration or invocation.
  - Around opening and closing square brackets in an array declaration or reference.

4.5 Braces and Line Breaks

Use (curly) braces, even for blocks with only one statement. This removes one common source of bugs and eases maintenance:

1. Statements can be inserted or removed within a block without worrying about adding or removing braces.
2. Reduces the problem of matching `else` clauses to `if` clauses.

EXAMPLE:

CORRECT

```
if (bottom < index) {
    this.topRow = index – this.rows;
} else if (index < this.topRow) {
    this.topRow = index;
}
```

INCORRECT

```
if (bottom < index)
    this.topRow = index – this.rows;
else if (index < this.topRow)
    this.topRow = index;
```

This rule applies to the following constructs:

- `for`, `while` and `do-while` loop.
- `if-else` statements.
- `try`, `catch` and `finally` clauses.
- `synchronized` blocks.

Note that the opening brace is at the end of the first line, even for class and method definitions. The only exception is if the expression needs to be broken; in that case, readability is best served by putting the opening brace on the next line.

4.6 Aligning Assignment Statements

- Align the "=" of related assignment statements. This sets them off as a group and shows clearly that they are related.
- Do not align the "=" of unrelated statements. Such alignment gives an impression that all statements are related.

EXAMPLE:

CORRECT

```
panelWidth  = 90;
panelHeight = 30;

selectedIndex = 0;
lastIndex     = 12;
```

INCORRECT

```
panelWidth    = 90;
panelHeight   = 30;
selectedIndex = 0;
lastIndex     = 12;
```

4.7 Line Lengths and Line Breaks

Avoid lines longer than 80 characters, since they are not handled well by many terminals and tools.

Break expressions that will not fit on a single line according to these general principles:

- Break after a comma.
- Break before an operator.
- Prefer higher-level breaks to lower-level breaks.
- Align the new line with the beginning of the expression at the same level on the previous line.
- If the above rules lead to confusing code or if a large amount of code is lined up against the right margin indent 8 spaces instead.

The following are examples of breaking method calls.

EXAMPLE:

```
someMethod(longExpression1, longExpression2, longExpression3,
        longExpression4, longExpression5);

var = someMethod1(longExpression1,
                someMethod2(longExpression2,
                        longExpression3));
```

4.8 Arithmetic Expression Layout

Following are two examples of laying out a long arithmetic expression. The first layout is the recommended standard, because the break occurs outside the parenthesized expression, which is at a higher level.

EXAMPLE:

CORRECT

```
longName1 = longName2 * (longName3 + longName4 – longName5)
           + 4 * longname6;
```

INCORRECT

```
longName1 = longName2 * (longName3 + longName4
                        - longName5) + 4 * longname6;
```

4.9 if Statement Layout

Line wrapping for `if` statements should generally use the 8-space rule, since conventional (4 spaces) indentation makes seeing the body difficult.

EXAMPLE:

CORRECT

```
if ((condition1 && condition2)
        || (condition3 && condition4)
        || ! (condition5 && condition6)) {
    doSomethingAboutIt();
}

//OR USE THIS
if ((condition1 && condition2) || (condition3 && condition4)
        || ! (condition5 && condition6)) {
    doSomethingAboutIt();
}
```

INCORRECT

```
if ((condition1 && condition2)
    || (condition3 && condition4)
    ||!(condition5 && condition6)) { //BAD WRAPS
    doSomethingAboutIt();            //MAKE THIS LINE EASY TO MISS
}
```

4.10 Ternary Expression Layout

There are three recommended ways to format ternary expressions. Any of these layouts are acceptable.

EXAMPLE:

```
alpha = (aLongBooleanExpression) ? beta : gamma;

alpha = (aLongBooleanExpression) ? beta
                                 : gamma;

alpha = (aLongBooleanExpression)
        ? beta
        : gamma;
```

4.11 switch/case Layout

- Indent the `case` clauses with respect to the `switch` statement.
- Indent the statements that belong to a `switch`, one statement to a line.
- In the case of large, repetitive lists of cases, it may be better to do a table layout.

EXAMPLE:

```
switch (some_value) {
    case case1:  bla_bla[0] = value1; break;
    case case2:  bla_bla[0] = value2; break;
    case case3:  bla_bla[0] = value3; break;
    ...
}
```

4.12 Anonymous Classes

An anonymous class is a particular form of inner classes – an innovation of Java 1.1. Here are two cases of when an anonymous class may be used and examples of how to format each case.

Case 1:
If a class is used more than once, assign an instance to a variable.

EXAMPLE:

```
ActionListener actionListener = new ActionListener() {
    public void processActionEvent(ActionEvent e) {
        ...
    }
};

this.comboBox.addActionListener(actionListener);
this.button.addActionListener(actionListener);
```

Case 2:
More often than not, the anonymous class is a listener designed to handle events from one specific widget only. In this case, define and instantiate the class directly in the code.

EXAMPLE:

```
this.comboBox.addActionListener(new ActionListener() {
    public void processActionEvent(ActionEvent e) {
        ...
    }
} );
```

5 Naming Conventions

Naming conventions are used to make programs more understandable and easier to read. They can also give information about the function of the identifier. For example, whether it's a constant, package, or class—which can be helpful in understanding the code.

5.1 Package Names

The prefix of a unique package name is written in all-lowercase ASCII letters and should be one of the top-level domain names, currently com, edu, gov, mil, net, org, or one of the English two-letter codes identifying countries as specified in ISO Standard 3166, 1981. All custom-developed packages for SFA will utilize `gov.sfa` as their naming base

EXAMPLE:

```
com.sum.eng

gov.sfa.example
```

Note: Refer to Deliverable 4.1.5 – Internet Standards in the appendix for additional information.

5.2 Classes

- Class names should be nouns, in mixed case with the first letter of each internal word capitalized.
- Try to keep class names simple and descriptive.
- Use whole words—avoid acronyms and abbreviations (unless the abbreviation is much more widely used than the long form, such as URL or HTML).

EXAMPLE:

```
class FunctionPanel;
class Raster;
```

5.3 Interfaces

Interface names should be capitalized like class names.

EXAMPLE:

```
interface Alpha
interface ImageSprite
```

5.4 Variables

- Variables are in mixed case with a lowercase first letter. Internal words start with capital letters. Variable names should not start with underscore "_" or dollar sign "$" characters, even though both are allowed.

- Variable names should be short yet meaningful. The choice of a variable name should be designed to indicate to the casual observer the intent of its use. One-character variable names should be avoided except for temporary "throwaway" variables. Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.
- Variables should *not* use ownership prefixes like "my" or prefixes like i for integer, d for double, etc... (implying type in name). However, *always* refer to them using the explicit "`this.fieldName`" notation. Field names should always start with a lower case letter and should separate words using capital letters, not underscores.

EXAMPLE:

```
int                 i;
float               squareWidth;
```

5.5 Methods

- Method names are in proper case, with initial lower-case letter. If possible, construct method names that follow the action-object paradigm, i.e., `getAccount`, `printAll`. The recommended method name is `getSize` rather than `size`.
- Methods are called in the context of their class. As a result, it is not necessary to repeat the class name in method names. If the class `Customer` has a method to retrieve the customer's name, name this method `getName` rather than `getCustomerName`. When users of the class invoke this method, they write something like `customer.getName()`, which is preferable to `customer.getCustomerName()`.
- When calling a method from another method in the same class or in a subclass of it, always use the explicit "`this.methodName()`", to emphasize the calling of a method of this class.

EXAMPLE:

```
class Circle {
    ...
    int getRadius() {
        return this.radius;
    }
    int getArea() {
        return (this.getRadius() * this.getRadius()) * Math.PI;
    }
}
```

- Method parameters should be prefixed by one of the following: "a", "an" or "new", depending on the method semantic.

EXAMPLE:

```
class Person {
    ...
    void setName(String newName) {
        this.name = newName;
    }
    void setID(long newID) {
        this.ID = newID;
    }
    boolean isEqual(Person aPerson) {
        return (this.getID() == aPerson.getID());
    }
}
```

5.6 Constants

- The names of variables declared class constants and of ANSI constants should be all uppercase with words separated by underscores ("_"). (ANSI constants should be avoided, for ease of debugging.)

EXAMPLE:

```
public static final int DEFAULT_COLOR = Color.black;

private static final String DEFAULT_SERVER = ”\\LF3DEV01”;
```

- Javadoc comments are required for public, protected and package constants. (For additional information on Javadocs refer to the Section **8** - Comments**.**)

5.7 Exceptions

- Exception names follow class-naming conventions, with the additional requirement that the name end in Exception.

EXAMPLE:

```
* @exception ResourceNotFoundException. recoverable, try another resource
```

6 Coding Objectives

This section defines coding objectives and defines each objective. Use this table as a guide for resolving design and implementation issues. The relative importance of each objective should be based upon the requirements of the application being built.

DEFINITIONS OF OBJECTIVES:

Requirement	Definition
Correctness	Verifying the code works correctly.
Size	This does not refer to the number of source code lines, but to the total size of compiled code (the `.class` files). It also includes overhead imposed by non-functional data, e.g., strings used internally in the program. Traditionally, size also includes memory usage.
Speed	This includes both execution speed (as measured by CPU usage) and perceived responsiveness from the user's point of view. These are not necessarily the same thing.
Robustness	Tolerance towards erroneous input and other error conditions. This does not mean that a program or routine should accept bad data, but that it should recognize and be able to handle it.
Safety	Choose the implementation approach that is likely to result in the fewest development errors (bugs).
Testability	Easy to test.
Maintainability	Code that is easy to maintain typically has several characteristics: • It is easy to read and understand. • It is well encapsulated. This allows changes (updates or fixes) to be made with some confidence that it won't impact something else. • Documentation, including comments in the code, is in agreement with the code.
Simplicity	Self-describing.
Reusability	This can mean class or function reuse in the same project, or it can mean preparing for reuse on a later project. Designing for reuse typically has an overhead of around 50%, split among additional design time, additional documentation requirements, and additional testing. A good compromise is to choose a design that does not *preclude* reuse.

Requirement	Definition
Portability	The code is reusable across platforms. Coding for portability typically includes: • Using a cross-platform library • Using a subset of a language or library that is common and consistent across platforms • Isolating platform dependencies When migrating Java code across platforms, differences between Java Virtual Machine implementations, library implementations, and host GUIs need to be accounted for. As a consequence Java code must be tested on a number of different hardware platforms, operating systems, and Web browsers to ensure portability and compatibility.

7 Programming Style

This section covers layout conventions and coding principles.

7.1 Visibility

- The visibility of fields and classes should be as narrow as possible.
- If outside access is needed to such fields, use access methods (a.k.a. getters and setters).

EXAMPLE:

```
class Person {
    private String name;
    public void setName(String newName) {
        this.name = newName;
    }

    public String getName() {
        return this.name;
    }
    ...
}
```

Whenever possible, access fields using getters even inside methods in the same class. This greatly simplifies maintenance and readability.

EXAMPLE:

CORRECT

```
class Circle {
    ...
    int getRadius() {
        return this.radius;
    }
    int getArea() {
        return (this.getRadius() * this.getRadius()) * Math.PI;
    }
}
```

INCORRECT

```
class Circle {
    ...
    int getRadius() {
            return this.radius;
      }
      int getArea() {
            return (this.radius * this.radius) * Math.PI;
      }
}
```

7.2 Code Granularity (Method Size)

- A reasonable line count for a method depends on the complexity of the method. A module that consists of sequential statements can be longer than a method containing complex conditions and loops. If the sequential code is repetitive, such as an index-by-index array initialization, the method may be as long as necessary to perform the required function.
- In general, methods should not be more that 15 "real" lines long (that is, not counting blank lines added for readability and comments). Methods should never be longer than 50 "real" lines. If a method's implementations is too long, decompose it.
- A method should preferably do one single thing, and the method name should reflect this function accurately. If it does additional functionality, ensure that this is reflected in the method name. If this leads to a complicated or confusing method name, reconsider the structure of the code.

EXAMPLE:

If there is a function named `initPanelManagerAndReadAccountList`, it would probably be beneficial to split the code into methods named `initPanelManager` and `readAccountList`.

7.3 Variables

One declaration per line is recommended since it encourages commenting.

EXAMPLE:

CORRECT

```
    int level; // indentation level
    int size;  // size of table
```

The example above uses one space between the type and the identifier. Another acceptable alternative is to space out and align the code.

EXAMPLE:

CORRECT

```
     int          level;                  // indentation level
     int          size;                   // size of table
     Object       currentEntry;           // currently selected table entry
```

Both options are preferred over:

EXAMPLE:

INCORRECT

```
    int level, size;
```

Do not put different types on the same line.

EXAMPLE:

INCORRECT

```
    int lib,  libarray[];
```

7.3.1 Initialization

- All variables, including fields and class variables, should be initialized at the point of declaration. Although all Java declarations have default initialization values (0, null, false), initialize all variables explicitly.
- Java allows initialization of arrays using the same syntax as C and C++, by enclosing a comma-delimited set of values in braces. A comma after the final value is permissible: use this facility, as it makes for easier maintenance—it is easier to add additional values to or remove values from the end of the list.
- Java 1.1 allows initializer blocks among the declarations. An initializer block is a section of code enclosed in braces. There are two kinds of initializer blocks: static and instance.

  Static initializer blocks are executed the first time a class is instantiated. During static initialization (class initialization), the following steps are taken:
  1. Class initialization of the superclass is performed, unless it has previously been done.
  2. Static variables are initialized and static initializer blocks are executed. This happens in the order they are listed, from top to bottom. (Does not apply to: instance variables, instance initializer blocks and methods.)

  Note that static and instance initializer blocks are allowed in Java 1.1. Static initializer blocks are executed in order when the class is first instantiated; instance initializer blocks are executed in order after the superclass constructor runs, but before the class constructor runs.

  Instance initializer blocks are executed whenever a class is instantiated. During object initialization (instance initialization), things happen in the following order:
  1. If this is the first time the class is instantiated, all the class (static) initialization takes place.
  2. Enter a constructor. If a constructor is not specified, the compiler automatically supplies a default constructor with no arguments.
  3. The superclass constructor is called. If a constructor does not explicitly invoke a superclass constructor, the default (argument-less) superclass constructor is called.
  4. All instance variables are initialized and instance initializer blocks are executed. This happens in the order they are listed, from top to bottom. (Does not apply to class variables, class initializer blocks and methods.)

- Use initializer blocks to perform any initialization that can't be performed by direct variable initialization; put each initializer block immediately following the variable in question. In the examples below, note that the array can be initialized without using an initializer block, while the vector object requires one because of the calls to the `addElement` method.

EXAMPLE:

```
private Vector listAnything = new Vector();

{   // Instance initializer block
    listAnything.addElement(someObject);
    listAnything.addElement(anotherObject);
}

private static int[] multipliers = {
    5, 4, 3, 2, 7, 6, 5, 4, 3, 2,
};
```

```
private static Screen screen = new Screen();

static {
    // Static initializer block
    screen.setValue(someValue);
}
```

7.3.2 Referring to Class Variables and Methods

Avoid using an object to access a class (static) variable or method. Use a class name instead.

EXAMPLE:

CORRECT
classMethod(); AClass.classMethod();
INCORRECT
anObject.classMethod();

7.3.3 Variable Usage

Always use a variable for a single purpose.

EXAMPLE:

INCORRECT

```
int i;
...
for (i = 0; i < accountList.size(); ++ i) {
    ...
}
...

// Swap elements:
i = anyArray[0];
anyArray[0] = anyArray[1];
anyArray[1] = i;

...
```

The two uses of the variable `i` above have nothing to do with one another. Creating unique variables for each purpose of the code will improve the readability.

Do not use the assignment operator in a place where it can be easily confused with the equality operator.

EXAMPLE:

CORRECT

```
 if ((c++ = d++) != 0) {
     ...
```

```
}
```

INCORRECT

```
if (c++ = d++) {
    ...
}
```

Do not use embedded assignments in an attempt to improve run-time performance.

EXAMPLE:

CORRECT

```
a = b + c;
d = a + r;
```

INCORRECT

```
d = (a = b + c) + r;
```

7.4 Constants

Numerical constants (literals) should not be explicitly coded, except for -1, 0, and 1, which can appear in a `for` loop as counter values.

7.5 Straight-line Code

Straight-line code can be divided into two categories:

1. A sequence of statements that must be in a specific order: In this case, there are dependencies between statements; one statement must be executed before another for the program logic to work correctly.

   Here are a few simple guidelines:
   - Organize the code so that the dependencies are clear.
   - Name methods so that dependencies are obvious at their point of call.
   - Use method parameters or return values to make dependencies obvious.
   - Document unclear dependencies.

2. A sequence of statements whose order doesn't matter: In this case, the program will work correctly no matter what the order of statements.

   Organize the statements so that readers do not need to skip around to find required information:
   - Keep related statements together.
   - Localize references to variables, i.e., declare and initialize variables as close as possible to where they are used.

7.6 Conditionals

- Complex conditions can be hard to read and understand. One way to improve readability is by using additional `boolean` variables. In the first code fragment below the meaning of the test is straightforward; in the second, it is not as obvious.

EXAMPLE:

CORRECT

```
final boolean finished = (element < 0 || MAX_ELEMENTS < element);
final boolean repeatedEntry = (element == lastElement);

if (finished || repeatedEntry) {
    ...
}
```

INCORRECT

```
if (element < 0 || MAX_ELEMENTS < element ||
    element == lastElement )
{
    ...
}
```

This approach both simplifies and documents complex expressions, making it easier to program and maintain without errors.

- If a chain of `if-then` statements are required, code the most common cases first.
- Strive to minimize the number of branches in the code. Linear code is easier to test.
- Limit nesting to three levels.
- Compare `boolean` values to `true` or `false` implicitly, not explicitly.

EXAMPLE:

CORRECT

```
if (valid) {
    ...
}
```

```
if (!valid) {
    ...
}
```

INCORRECT

```
if (valid == true) {
    ...
}
```

```
if (valid == false) {
    ...
}
```

7.7 Loops

- It is recommended that a `for` loop is used whenever possible. The advantages of the `for` loop is that it collects the loop control in a single place and that it allows a loop control variable to be declared that is not accessible outside the loop.

EXAMPLE:

```
for (int i = 0; i < vector.size(); ++i ) {
    ...
}
```

- Never modify the loop control variable inside the `for` loop. If modifying the control variable becomes necessary, use a `while` loop instead.

EXAMPLE:

CORRECT

```
int i = 0;
while (i < vector.size()) {
    Person person = (Person) vector.elementAt(i);
    if (person. isTired()) {
        vector.removeElementAt(i);
    } else {
        ++ i;
    }
}
```

INCORRECT

```
for (int i = 0; i < vector.size(); ++ i ) {
    Person person = (Person) vector.elementAt(i);
    if (person.isTired()) {
        vector.removeElementAt(i);
        -- i; // Loop control is off limits!
    }
}
```

- Use loops that test exit conditions at the top or the bottom. If this cannot be easily accomplished, rewrite the loop as a “`while (true)`” infinite loop with a test in the middle of the loop. If possible, use only a single break statement to exit the loop.
- If possible, make loops short enough to view all at once. This is especially important if the loop body is complex. If the loop code grows beyond about 15 lines, consider restructuring the code.
- Limit nesting to three levels.

7.8 Switches

- Never let flow control ”fall through” from one case label to the next by omitting the break statement.

7.9 Notes on Specific Usage and Constructs

7.9.1 final

Apply the `final` keyword to classes, methods and variables:

- A final class may not be subclassed.
- A final method may not be overridden.
- A final variable may never be changed.

Using `final` on a class, method, or variable may have an optimization effect on the Java code. The compiler may be able to perform inlining or compile-time linking instead of dynamic linking at run-time. For this reason, apply `final` to all classes and methods that are not intended to be subclassed and overridden.

7.9.2 Constructors

- There should normally be only one "main" constructor in a class. Additional convenience constructors may be defined, but they should be implemented in terms of the main constructor. This will reduce duplicate code.

EXAMPLE:

"MAIN" CONSTRUCTOR

```
public MultiLineLabel(
    String aLabel,
    int aMarginWidth,
    int aMarginHeight,
    int aTextAlignment,
    int aFixedSize)
{
    this.breakLabel(aLabel);
    this.marginWidth  = aMarginWidth;
    this.marginHeight = aMarginHeight;
    this.textAlignment = aTextAlignment;
    this.fixedWidth = aFixedSize;
}
```

WRONG CONVENIENCE CONSTRUCTOR (REPEATS CODE FROM ABOVE)

```
public MultiLineLabel(String aLabel) {
    this.breakLabel(aLabel);
    this.marginWidth  = 0;
    this.marginHeight = 0;
    this.textAlignment = Alignment.LEFT;
    this.fixedWidth = 0;
}
```

CORRECT CONVENIENCE CONSTRUCTOR

```
public MultiLineLabel(String aLabel) {
    this(aLabel, 0, 0, Alignment.LEFT, 0);
}
```

7.9.3 Threads

Debugging and profiling can be more effective by naming all threads explicitly. Therefore, use the `Thread` constructors that take a name parameter, (e.g. use `Thread(String aName)` instead of `Thread()`.)

8 Comments

The two main comment types are implementation comments and documentation comments.

Implementation comments are those found in C++, which are delimited by /*...*/, and //. Implementation comments are meant for commenting out code or for comments about a particular implementation.

Documentation comments (known as "Javadoc comments") are Java-only, and are delimited by /**...*/. Document comments can be extracted to HTML files using the Javadoc tool. Document comments are meant to describe the specification of the code, from an implementation-free perspective and are to be read by developers who might not have the source code at hand.

Comments should be used to give overviews of code and provide additional information that is not readily available in the code itself. Comments should contain only information that is relevant to reading and understanding the program. For example, information about how the corresponding package is built or in what directory it resides should not be included as a comment.

Comments should not be enclosed in large boxes drawn with asterisks or other characters.

Comments should never include special characters such as form-feed and backspace.

8.1 Implementation Comment Formats

Programs can have four styles of implementation comments: block, single-line, trailing, and end-of-line.

8.1.1 Block Comments

Block comments are used to provide descriptions of files, methods, data structures and algorithms. Block comments may be used at the beginning of each file and before each method. They can also be used in other places, such as within methods. Block comments inside a function or method should be indented to the same level as the code they describe.

A block comment should be preceded by a blank line to set it apart from the rest of the code.

EXAMPLE:

```
/*
 * Here is a block comment.
 */
```

8.1.2 Single-Line Comments

Short comments can appear on a single line indented to the level of the code that follows. If a comment can't be written in a single line, it should follow the block comment format (see section **8.1.1**). A single-line comment should be preceded by a blank line. Here's an example of a single-line comment in Java code (also see "Documentation Comments" – section **8.2**).

EXAMPLE:

```
if (condition) {

    /* Handle the condition. */
    ...
}
```

8.1.3 Trailing Comments

Short comments can appear on the same line as the code they describe, but should be shifted far enough to separate them from the statements. If more than one short comment appears in a chunk of code, they should all be indented to the same tab setting.

EXAMPLE:

```
if (a == 2) {
    return TRUE;            /* special case */
} else {
    return isPrime(a);      /* works only for odd a */
}
```

8.1.4 End-Of-Line Comments

The // comment delimiter can comment out a complete line or only a partial line. It shouldn't be used on consecutive multiple lines for text comments; however, it can be used in consecutive multiple lines for commenting out sections of code.

EXAMPLES OF ALL THREE STYLES FOLLOW:

```
if (any > 1) {

    // Do a double-flip.
    ...
}
else {
    return false;          // Explain why here.
}
//if (thi > 1) {
//
//    // Do a triple-flip.
//    ...
//}
//else {
//    return false;
//}
```

8.2 Documentation Comments

Note: See "Java Source File Example" for examples of the comment formats described here.

For further details, see "How to Write Doc Comments for Javadoc" which includes information on the Javadoc comment tags (@return, @param, @see):

http://java.sun.com/products/jdk/javadoc/writingdoccomments.html

For further details about Javadoc comments and Javadoc, see the Javadoc home page at:

http://java.sun.com/products/jdk/javadoc/

Javadoc comments describe Java classes, interfaces, constructors, methods, and fields. Each Javadoc comment is set inside the comment delimiters /**...*/, with one comment per class, interface, or member. This comment should appear just before the declaration.

EXAMPLE:

```
/**
 * The Example class provides ...
 */
public class Example { ...
```

Notice that top-level classes and interfaces are not indented, while their members are. The first line of Javadoc comment (/**) for classes and interfaces is not indented; subsequent Javadoc comment lines each have one space of indentation (to vertically align the asterisks). Members, including constructors, have four spaces for the first Javadoc comment line and five spaces thereafter.

If there is a need to provide information about a class, interface, variable, or method that isn't appropriate for documentation, use an implementation block comment (see section **8.1.1**) or single-line (see section **8.1.2**) comment immediately after the declaration. For example, details about the implementation of a class should go in such an implementation block comment following the class statement, not in the class Javadoc comment.

Doc comments should not be positioned inside a method or constructor definition block, because Java associates documentation comments with the first declaration after the comment.

9 Code Examples

The following example shows how to format a Java source file containing a single public class. Interfaces are formatted similarly.

```
/*
 * @(#)Sfa.java
 * 1.82
 * 99/03/18
 * Jane Doe – SFA
 *
 *
 *
 */


package java.any;

import java.any.anyt.AnyAny;

/**
 * Class description goes here.
 *
 * @version     1.82 18 Mar 1999
 * @author      Firstname Lastname
 */
public class Any extends SomeClass {
    /* A class implementation comment can go here. */

    /** classVar1 documentation comment */
    public static int classVar1;

    /**
     * classVar2 documentation comment that happens to be
     * more than one line long
     */
    private static Object classVar2;

    /** instanceVar1 documentation comment */
    public Object instanceVar1;

    /** instanceVar2 documentation comment */
    protected int instanceVar2;

    /** instanceVar3 documentation comment */
    private Object[] instanceVar3;

    /**
     * ...constructor Any documentation comment...
     */
    public Any() {
        // ...implementation goes here...
    }

    /**
     * ...method doSomething documentation comment...
     */
    public void doSomething() {
        // ...implementation goes here...
    }
```

```
    /**
     * ...method doSomethingElse documentation comment...
     * @param someParam description
     */
    public void doSomethingElse(Object someParam) {
        // ...implementation goes here...
    }
}
```

10 JDBC Standards

10.1 JDBC Overview

Java Database Connectivity (JDBC) is the current standard for Java programs to access information from a database. JDBC is a "low-level" database independent API that enables the creation of platform independent client/server database applications. It does this by utilizing database specific drivers. Similar to ODBC (Open Database Connectivity), JDBC is also based on the X/Open SQL Call-Level Interface.

JDBC is used for 3 specific purposes:

- connecting to a database.
- executing SQL statements.
- processing the results.

10.2 JDBC Programming Overview

Initially a developer will load the JDBC classes to the Java application or applet class. Then in order to incorporate database calls through a Java application, applet, or servlet to any database, the developer must identify the name of the JDBC driver to be used and include it in the distribution of the application.

Within the code, there are four major steps in the implementation of JDBC:

1. Create the connection between the client and the database.

2. Create statement handles for passing data from the database into variables in the Java program.

3. Manipulate the variables via the programming logic.

4. Move data back into the database before closing out the connection and statements. (As Needed)

10.3 JDBC Drivers Overview

JDBC Drivers are a set of classes that implement the JDBC interfaces for a particular database. There are four main types of drivers:

10.3.1 JDBC-ODBC Drivers (Type 1)

The JDBC-ODBC drivers translate JDBC method calls into ODBC function calls. This enables a client to connect to an ODBC database via Java calls and JDBC, which means that the neither the database nor middle tier needs to be Java compliant. The main advantage is that applications can easily access databases from multiple vendors by choosing the an appropriate ODBC driver. However, when ODBC is used, the ODBC driver manager and drivers must be installed on every client that uses the ODBC APIs. Figure 1 shows how JDBC accesses a database via a Type 1 Driver:

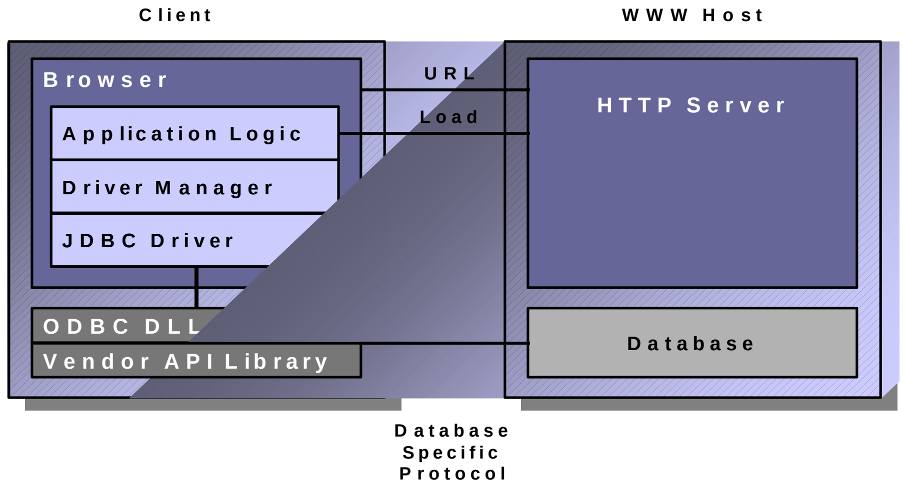

Figure 1: JDBC Type 1 Driver

10.3.2 Native-API, Partly Java Drivers (Type 2)

Native-API, Partly Java Drivers convert JDBC calls into calls for a specific database, which requires a vendor-supplied library to translate JDBC functions into the DBMS's specific query language. This driver also requires a native vendor-supplied library stored on the client, which is not suitable for downloading over a network. These Type 2 drivers are as not as flexible with database platforms as the Type 1 drivers, but Type 2 drivers are faster due to the ODBC translation layer being removed. Figure 2 identifies how JDBC accesses a database via a Type 2 Driver:

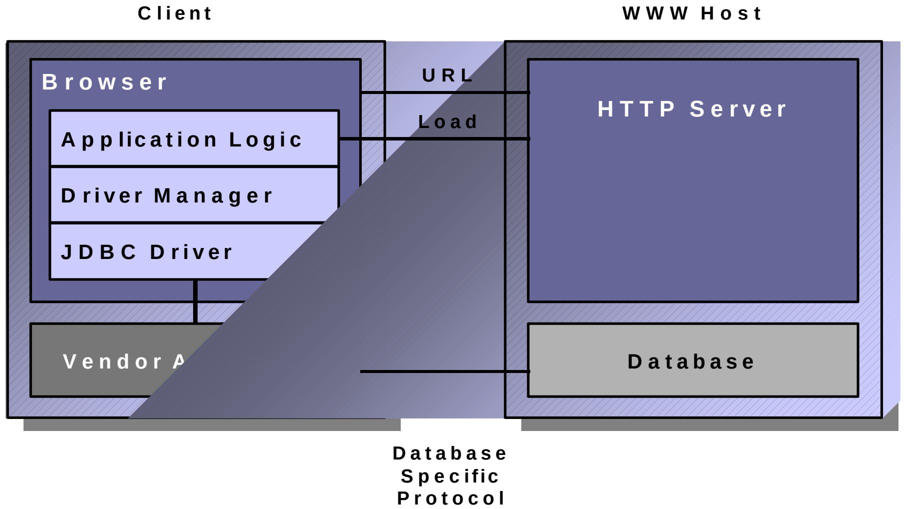

Figure 2: JDBC Type 2 Driver

10.3.3 JDBC-Net Pure Java Drivers (Type 3)

JDBC-Net Pure Java Drivers convert JDBC calls into a database-independent net protocol, which are translated to database specific APIs by a middle tier server. The overall architecture consists of three tiers: the JDBC client and driver, middleware (software that sits on a middle tier between an object residing on one server machine and any clients that want to access that object), and the database(s) being accessed. Type III drivers are best suited for Internet/intranet-based, multi-user data-intensive applications, including concurrent data operations where scalability and performance is a major requirement. The server can handle multiplexing management among several databases, provide logging and administration facilities, load balancing features, and support catalog and query caches. Figure 3 identifies how JDBC accesses a database via a Type 3 Driver:

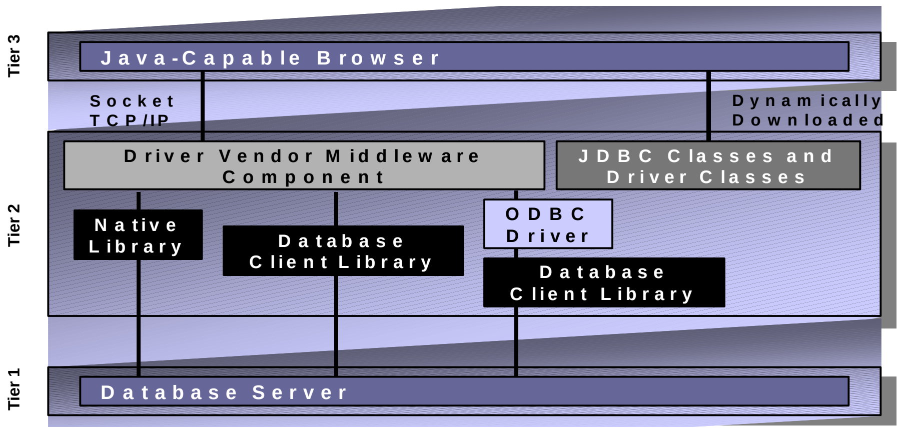

Figure 3: JDBC Type 3 Driver

10.3.4 Native-Protocol, Pure-Java Drivers (Type 4)

Native-Protocol, Pure-Java Drivers convert JDBC calls directly into proprietary database vendor protocols without the use of APIs. They can be written entirely in Java and (like Type 3 drivers) can provide for just-in-time delivery of applets, which provides high performance. Type 4 drivers are only available from the DBMS vendors. Figure 4 identifies how JDBC accesses a database via a Type 4 Driver:

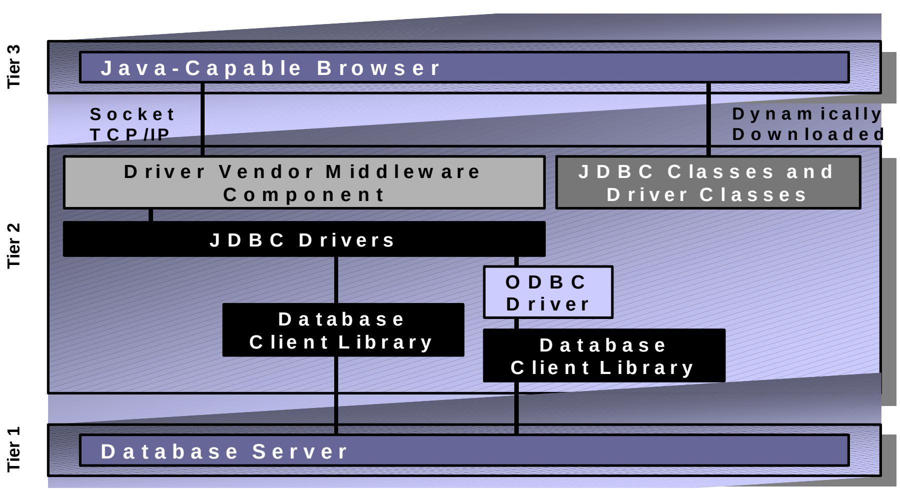

Figure 4: JDBC Type 4 Driver

11 Visual Age for Java

IBM's Visual Age for Java, Enterprise Edition, version 3.0 is the current standard SFA Java development tool. The software provides support for the build, test and deployment stages of Java applications, JavaBeans components, servlets and applets. Visual Age for Java also provides tools that support a high-level of code reuse and allows for integrated team development.

The main components of the Visual Age suite include:

- *Integrated Development Environment* – set of windows which include the Workbench, Log, Console, Debugger, and Repository Explorer. These windows provide access to the development tools to create Java applets and standalone Java applications.

- *Visual Composition Editor (VCE)* – used to create object-oriented programs by manipulating graphics (visual programming). Specifically, graphical user interfaces are created from prefabricated beans and relationships are defined (called connections) between beans.

- *Team Programming Support* – provides a team development environment through a shared code repository.

- **Builders**

  – *Enterprise Access Builder for Transactions (EAB)* – uses connectors that are based on IBM's Common Connector Framework (CCF) to provide interfaces to a number of environments, including CICS, MQSeries.

  – *RMI Access Builder* – provides remote access to Java beans in a distributed, client-server environment.

  – *C++ Access Builder* – provides access from Java applications to services written in C++.

- *CICS Transaction Server* – allows the creation of Java applications that execute under CICS Transaction Server for OS / 390 Release 3.1.

- **Data Access**

  – *DB2 Stored Procedure Builder (SPB)* – enables the writing of Java stored procedures to run on a DB2 server.

  – *SQLJ* – allows the embedding of SQL statements in a Java program with embedded SQLJ, a translator, and a runtime environment.

  – *Enterprise Access Builder for Persistence (Persistence Builder)* – enables the mapping of information stored in relational databases to objects and relationships between objects.

- *Distributed Debugger* – is a client server application which enables the debugging of Java applications.

- *Enterprise Java Bean (EJB) Development Environment* – enables the development and testing of EJBs and access (adapter) beans. It also provides an incremental consistency checker to ensure EJBS are written to programming specifications.

- **Enterprise Toolkits:**

  – *Enterprise Toolkit for Workstations (ET/Workstation)* – enables the development of platform-specific code on Windows NT.

  – *Enterprise Toolkit for OS/390 (ET/390)* – allows the development of platform-specific code for the OS / 390 platform.

- *External SCM Tools* – allows the use of an external software configuration management (SCM) system form within Visual Age for Java.

- *IDL Development Environment* – allows the work of both the integrated interface definition language (IDL) and generated Java code in one browser page that works with a user-specified IDL-to-Java compiler.

- *JSP/Servlet Development Environment* – incorporates the development and testing of Java Server pages.

- *Migration Assistant* - creates a code framework to migrate Active X controls to Java Beans.

- *Tivoli Connection* – generates Tivoli events and interfaces to the Tivoli Enterprise Console to manage your Java applications.

- *Tool Integrator* – integrates Java applications that reside on the file system in order to launch them from within the Integrated Development Environment (IDE).

- *XML Metadata Interchange (XMI) toolkit* – integrates with the Rational Rose modeling tool. Java converts the Rose model (.mdl files) into an XMI (the XML data type definition for UML) format which facilitates the rapid transformation of business models.

12 Glossary

Term	Definition
Abstract base class	A class from which no objects may be created; it is only used as a base class for the derivation of other classes. A class is abstract if it includes at least one member function that is declared as *abstract.*
Accessor	A method which returns the value of a data member.
Catch clause	Code that is executed when an exception of a given type is raised. The definition of an exception handler begins with the keyword catch.
Class	A user-defined data type which consists of data elements and methods which operate on that data. Data defined in a class is called member data and methods defined in a class are called *member functions* or *methods.*
Constructor	A method which initializes an object.
Default constructor	A constructor which needs no arguments.
Default visibility members of a class	Member data and functions which are accessible within the package by specifying an instance of the class (or one of its subclasses) and the name. They are not accessible from outside the package.
Exception	A run-time program anomaly that is detected in a function or member function. Exception handling provides for the uniform management of exceptions. When an exception is detected, it is *thrown* (using a throw expression) to the exception handler.
Finally clause	Code that is executed at the end of an exception handling *try* block regardless of how the block is exited.
Forwarding method	A method which does nothing more than call another method.
Identifier	A name which is used to refer to a variable, constant, method or type in Java. When necessary, an identifier may have an internal structure which consists of a prefix, a name, and a suffix (in that order).

Term	Definition
Immutables	Those classes or objects whose state can not be changed, such as strings or final variables. In the case of *immutable* objects, their state is determined solely by the constructor.
Iterator	An object which, when invoked, returns the next object from a collection of objects.
Mutables	Those object which are not immutable or primitive.
Overloaded method name	A name which is used for two or more methods or member functions having different arguments.
Overridden member function	A member function in a base class which is re-defined in a derived class.
Pre-defined data type or primitive	A type which is defined in the language itself, such as int.
Private members of a class	Only visible within member functions of the class itself.
Protected members of a class	Member data and functions which are accessible within the package by specifying an instance of the class (or one of its subclasses) and the name. Outside the package these members are only accessible from member functions within derived classes
Public members of a class	Member data and member functions which are accessible everywhere by specifying an instance of the class and the name.
Reference	A pointer to an object. Any expression which uses “new” to create an object returns a reference. When an object is passed as a parameter, or returned by a method a new reference is created. When a primitive (e.g., an integer or character) is passed in or out of a method, a copy of the object is made and a new reference is not needed
Scope of a name	Refers to the context in which it is visible.[1]
Sliced	When an object that has been converted to a superclass object and some of its data has become inaccessible.

[1] Context, here, means the methods or blocks in which a given variable name can be used.

13 References

Code Conventions for the Java Programming Language – Sun Microsystems
http://java.sun.com/docs/codeconv/html/CodeVonvTOC.doc.html

Examining JDBC Drivers, by Mukul Sood
Dr. Dobb's Journal January 1998
http://www.ddj.com/articles/1998/9801/9801i/9801i.htm

IBM's About VisualAge for Java, Enterprise Edition, Version 3.0
http://www-4.ibm.com/software/ad/vajava/vaj3enterprise.html

Java Coding Standards – Andersen Consulting
Resources eCommerce Methodology

JDBC Data Access API - Sun
http://java.sun.com/products/jdbc/

Netscape's Software Coding Standards Guide for Java
http://developer.netscape.com/docs/technote/java/codestyle.html

Oracle's Introduction to JDBC
http://www.oracle.com/oramag/webcolumns/tbrief.html

Oracle Java Roadmap: Java Overview
http://technet.oracle.com/tech/java/jroadmap/java/listing.htm

14 Appendix A – Deliverable 4.1.5